Appendix A Mathematical Background A.1 Complexity Analysis and O() Notation Computer scientists are often faced with the task of comparing algorithms to see how fast they run or how much memory they require. There are two approaches to this task. The first is benchmarking—running the algorithms on a computer and measuring speed in seconds and memory consumption in bytes. Ultimately, this is what really matters, but a benchmark can be unsatisfactory because it is so specific: it measures the performance of a particular program written in a particular language, running on a particular computer, with a particular compiler and particular input data. From the single result that the benchmark provides, it can be difficult to predict how well the algorithm would do on a different compiler, computer, or data set. The second approach relies on a mathematical analysis of algorithms, independent of the particular implementation and input, as discussed below. Benchmarking Analysis of algorithms A.1.1 Asymptotic analysis We will consider algorithm analysis through the following example, a program to compute the sum of a sequence of numbers:function SUMMATION(sequence) returns a number for to LENGTH(sequence) do return sum The first step in the analysis is to abstract over the input, in order to find some parameter or parameters that characterize the size of the input. In this example, the input can be characterized by the length of the sequence, which we will call . The second step is to abstract over the implementation, to find some measure that reflects the running time of the algorithm but is not tied to a particular compiler or computer. For the SUMMATION program, this could be just the number of lines of code executed, or it could be more detailed, measuring the number of additions, assignments, array references, and branches executed by the algorithm. Either way gives us a characterization of the total number of steps taken by the algorithm as a function of the size of the input. We will call this characterization . If we count lines of code, we have for our example. If all programs were as simple as SUMMATION, the analysis of algorithms would be a trivial field. But two problems make it more complicated. First, it is rare to find a parameter like that completely characterizes the number of steps taken by an algorithm. Instead, the best we can usually do is compute the worst case or the average case . Computing an average means that the analyst must assume some distribution of inputs. The second problem is that algorithms tend to resist exact analysis. In that case, it is necessary to fall back on an approximation. We say that the SUMMATION algorithm is , meaning that its measure is at most a constant times , with the possible exception of a few small values of . More formally, The notation gives us what is called an asymptotic analysis. We can say without question that, as asymptotically approaches infinity, an algorithm is better than an algorithm. A single benchmark figure could not substantiate such a claim. sum ← 0 i = 1 sum ← sum + sequence [i] n T(n) T(n) = 2n + 2 n Tworst(n) Tavg(n) O(n) n n T (n) is O (f (n)) if T (n) ≤ kf (n) for some k, for all n > n0 . O() n O(n) O(n 2)Asymptotic analysis The notation abstracts over constant factors, which makes it easier to use, but less precise, than the notation. For example, an algorithm will always be worse than an in the long run, but if the two algorithms are and , then the algorithm is actually better for . Despite this drawback, asymptotic analysis is the most widely used tool for analyzing algorithms. It is precisely because the analysis abstracts over both the exact number of operations (by ignoring the constant factor) and the exact content of the input (by considering only its size) that the analysis becomes mathematically feasible. The notation is a good compromise between precision and ease of analysis. A.1.2 NP and inherently hard problems The analysis of algorithms and the notation allow us to talk about the efficiency of a particular algorithm. However, they have nothing to say about whether there could be a better algorithm for the problem at hand. The field of complexity analysis analyzes problems rather than algorithms. The first gross division is between problems that can be solved in polynomial time and problems that cannot be solved in polynomial time, no matter what algorithm is used. The class of polynomial problems—those which can be solved in time for some —is called P. These are sometimes called “easy” problems, because the class contains those problems with running times like and . But it also contains those with time , so the name “easy” should not be taken too literally.

Complexity analysis P O() T() O(n 2) O(n) T(n 2 + 1) T(100n + 1000) O(n 2) n < 110 k n O() O() O(n k) k O(log n) O(n) O(n 1000)Another important class of problems is NP, the class of nondeterministic polynomial problems. A problem is in this class if there is some algorithm that can guess a solution and then verify whether a guess is correct in polynomial time. The idea is that if you have an arbitrarily large number of processors so that you can try all the guesses at once, or if you are very lucky and always guess right the first time, then the NP problems become P problems. One of the biggest open questions in computer science is whether the class NP is equivalent to the class P when one does not have the luxury of an infinite number of processors or omniscient guessing. Most computer scientists are convinced that that NP problems are inherently hard and have no polynomial-time algorithms. But this has never been proven. NP NP-complete Those who are interested in deciding whether look at a subclass of NP called the NP-complete problems. The word "complete" is used here in the sense of "most extreme" and thus refers to the hardest problems in the class NP. It has been proven that either all the NP-complete problems are in P or none of them is. This makes the class theoretically interesting, but the class is also of practical interest because many important problems are known to be NP-complete. An example is the satisfiability problem: given a sentence of propositional logic, is there an assignment of truth values to the proposition symbols of the sentence that makes it true? Unless a miracle occurs and there can be no algorithm that solves all satisfiability problems in polynomial time. However, AI is more interested in whether there are algorithms that perform efficiently on typical problems drawn from a pre determined distribution; as we saw in Chapter 7 , there are algorithms such as WALKSAT that do quite well on many problems. The class of NP-hard problems consists of those problems that are reducible (in polynomial time) to all the problems in NP, so if you solved any NP-hard problem, you could solve all P ≠ NP; P = NP P = NP, the problems in NP. The NP-complete problems are all NP-hard, but there are some NP hard problems that are even harder than NP-complete. NP-hard The class co-NP is the complement of NP, in the sense that, for every decision problem in NP, there is a corresponding problem in co-NP with the "yes" and "no" answers reversed. We know that P is a subset of both NP and co-NP, and it is believed that there are problems in co-NP that are not in P. The co-NP-complete problems are the hardest problems in co NP. Co-NP Co-NP-complete The class #P (pronounced "number P" according to Garey and Johnson (1979), but often pronounced "sharp P") is the set of counting problems corresponding to the decision problems in NP. Decision problems have a yes-or-no answer: is there a solution to this 3- SAT formula? Counting problems have an integer answer: how many solutions are there to this 3-SAT formula? In some cases, the counting problem is much harder than the decision problem. For example, deciding whether a bipartite graph has a perfect matching can be done in time (where the graph has vertices and edges), but the counting problem "how many perfect matches does this bipartite graph have" is #P-complete, meaning that it is hard as any problem in #P and thus at least as hard as any NP problem. Another class is the class of PSPACE problems—those that require a polynomial amount of space, even on a nondeterministic machine. It is believed that PSPACE-hard problems are O(V E) V Eworse than NP-complete problems, although it could turn out that just as it could turn out that NP = PSPACE, P = NP.